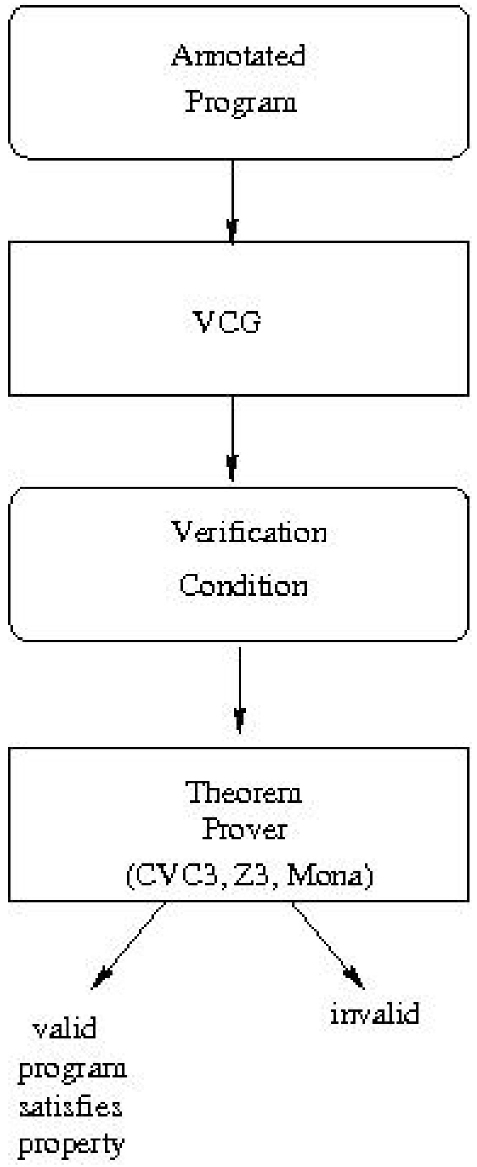
Lecture 3

Plan

- Verification Condition Concept
- Demo: proving formulas in Princess
- Language of Guarded Commands
- Programs as relations

Verification-Condition Generation


Annotated Program
VCG
Verification Condition
Theorem Prover (CVC3, Z3, Mona)
valid program satisfies property
invalid

Steps in Verification

- generate a **formulas whose validity implies correctness** of the program
- attempt to prove all formulas
  - if formulas all valid, program is correct
  - if a formula has a counterexample, it indicates one of these:
    - error in the program
    - error in the property
    - error in auxiliary statements (e.g. loop invariants)

Terminology

- generated formulas:
  verification conditions
- generation process:
  verification-condition generation
- program that generates formulas:
  verification-condition generator (VCG)

Program from Before

```
res = 0
i = x
while invariant res + 2*i == 2*x
  (i > 0) {
   i = i - 1
   res = res + 2
}
assert(res == 2*x)
```

VC for invariant preservation (proved it by hand):

$$res + 2*i = 2*x \ \wedge\ i_1=i - 1 \ \wedge\ res_1=res+2 \rightarrow res_1 + 2*i_1 = 2*x$$

This VC in the Princess Prover

```
res = 0
i = x
while invariant res + 2*i == 2*x
  (i > 0) {
   i = i - 1
   res = res + 2
}
assert(res == 2*x)
```

```
\universalConstants {
    int x, i, res, i1, res1; }
\problem {
(res + 2*i = 2*x & i1 = i - 1 & res1 = res + 2)
  -> res1 + 2*i1 = 2*x }
```

A More Difficult Example

$\exists$ x,y,k,p.

$(x < y + 2 \land y < x + 1 \land x = 3k \land$

$(y = 6p+1 \lor y = 6p-1))$

Is this statement true? Or, replace y+2 with y+1.

Is a formula of **Presburger arithmetic** satisfiable?

$\mathbf{F ::= A \mid F_1 \land F_2 \mid F_1 \lor F_2 \mid \neg F \mid \exists k.F \mid \forall k.F}$

$\mathbf{A ::= T_1 = T_2 \mid T_1 < T_2}$

$\mathbf{T ::= k \mid C \mid T_1 + T_2 \mid T_1 - T_2 \mid C * T \mid T \;\%\; C}$

In Princess

```
\existentialConstants {
  int x,y,k,p;
}
\problem {
     (x < y + 2 & y < x + 1 & x = 3 * k
   & (y = 6 * p + 1 | y = 6*p - 1))
}
```

Remarks:

- Existential constants are like existentially quantified variables
- If all symbols are quantified (like x,y,k,p) or have a fixed interpretation (like *, +, =, <) then we satisfiability and validity are the same, and we can talk simply whether the formula is true or false.

Simple Programming Language

x = T
if (F) c1 else c2
c1 ; c2
while (F) c1

ordinary control structures

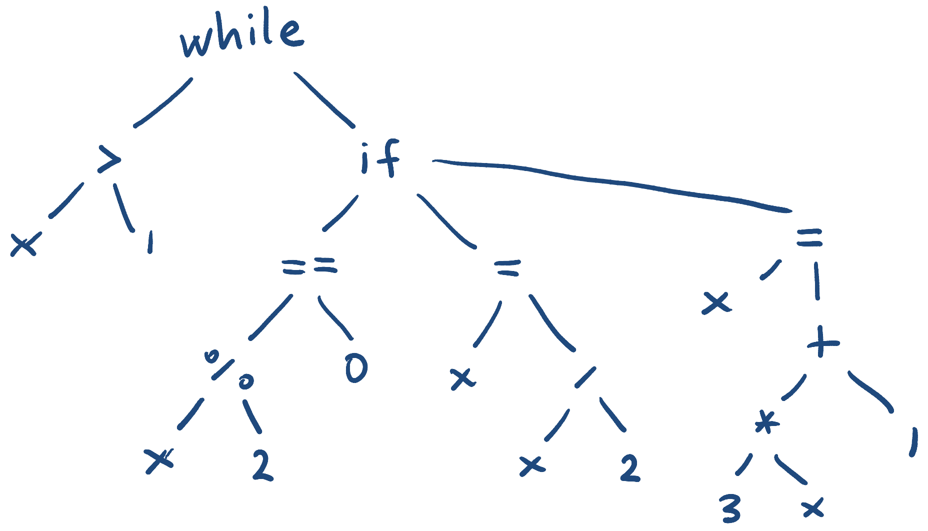
terms like in P.A.

c ::= x=T | (if (F) c else c) | c ; c | (while (F) c)
T ::= K | V | (T + T) | (T - T) | (K * T) | (T / K) | (T % K)
F ::= (T==T) | (T < T) | (T > T) | (~F) | (F && F) | (F || F)
V ::= x | y | z | ...
K ::= 0 | 1 | 2 | ...

Boolean terms like
P.A. formulas without quantifiers

Simple Program and its Syntax Tree

```
while (x > 1) {
  if (x % 2 = 0)
    x = x / 2
  else
    x = 3 * x + 1
}
```



All programs are integers, and are initially zero.

Remark: Turing-Completeness

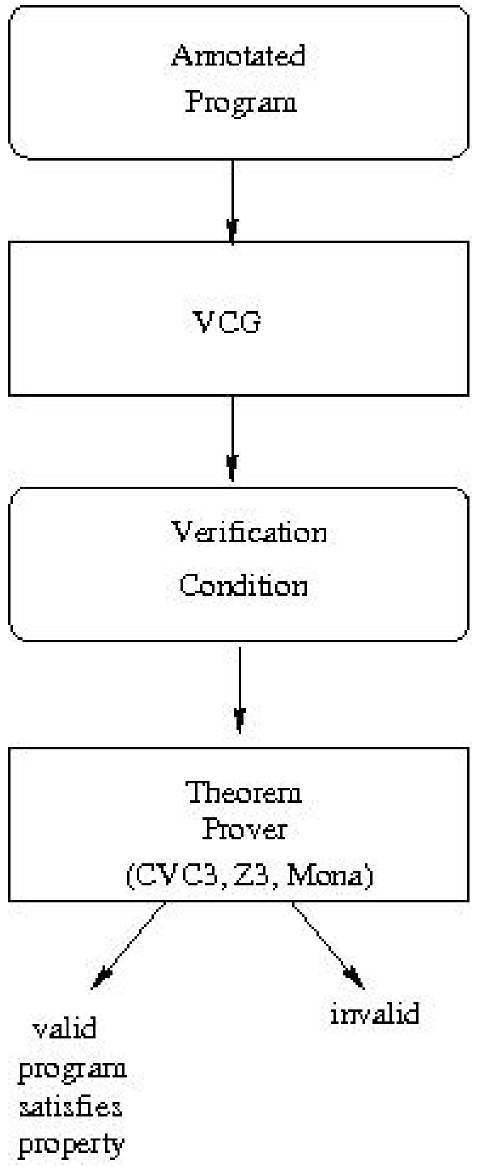
This language is Turing-complete
- it subsumes counter machines, which are known to be Turing-complete
- every possible program (Turing machine) can be encoded into computation on integers (computed integers can become very large)
- the problem of taking a program and checking whether it terminates is undecidable
- Rice's theorem: all properties of programs that are expressed in terms of the results that the programs compute
(and not in terms of the structure of programs) are undecidable

In real programming languages we have bounded integers, but we have other sources of unboundedness, e.g.
- BigInt data type of Java and Scala (sequence of digits of any length)
- example: sizes of linked lists and of other data structure
- program syntax trees for an interpreter or compiler
  (we would like to handle programs of any size!)

What is decidable

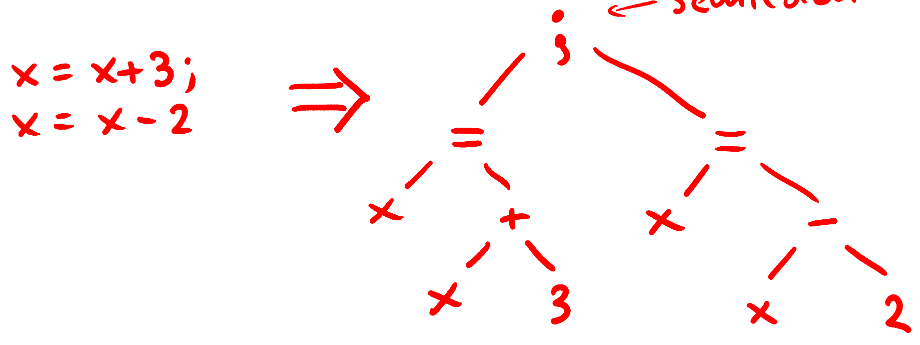
- Checking satisfiability of Presburger arithmetic formulas (even with quantifiers) is decidable
- Checking if there exists an input to a program in our language for which program computes a given value (e.g. 1) is undecidable
- Quantifiers in Presburger arithmetic cannot be used to define z=x*y, but we can write a program that computes x*z and stores it in z
- Programs without loops can be translated into Presburger arithmetic
- Loops give much more expressive power to Presburger arithmetic than quantifiers (situation can be different if we did not work with Presburger arithmetic)

Relational Semantics


Annotated Program
VCG
Verification Condition
Theorem Prover (CVC3, Z3, Mona)
valid program satisfies property
invalid

relation
(infinite mathematical object)

given by set comprehension
(formula, with finite syntax tree)

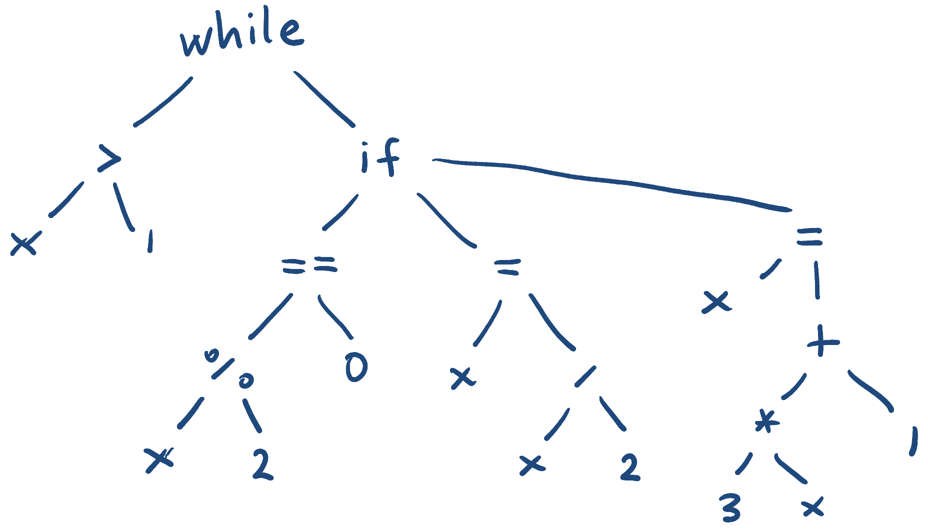


⇓

$\{(x,x') \mid x' = x+1\} \Leftarrow \{(0,1), (1,2), (2,3), (3,4), \ldots\}$

Mapping Programs

```
while (x > 1) {
  if (x % 2 = 0)
    x = x / 2
  else
    x = 3 * x + 1
}
```



into relations

$$\{((x_1,\ldots,x_n), (x'_1,\ldots,x'_n)) \mid F(x_1,\ldots,x_n, x'_1,\ldots,x'_n)\}$$

initial state final state

Examples

```
x = x+3;
x = x+2
```

$\{(x,x') \mid x' = x+5\}$

```
x = x+x
```

$\{(x,x') \mid x' = 2x\}$

```
while (x != 10)
   x = x+1
}
```

$\{(x,x') \mid x \leq 10 \wedge x' = 10\}$

```
while (5==5) {
   x = x
}
```

$\emptyset$

Relation between initial and all possible final states.

Example of Non-Determinism

```
x = randomInteger()
if (x > 10) {
  y = y+1
} else {
  y = y+2
}
```

relation between the initial and the final y:

$\{(y,y') \mid (y' = y+1 \;\|\; y' = y + 2)\} =$

$\{\ldots,(100,101),(100,102), (101,102),(101,103), \ldots\}$

obviously, not a function

Why Relations

The meaning is, in general, an arbitrary *relation*. Therefore:

• For certain states there will be **no results**.
  In particular, if a computation starting at a state does not terminate
  (due to a program that blocks on input, or loops forever)

• For certain states there will be **multiple results**.
  This means command execution starting in state will sometimes
compute one and sometimes other result.
  Verification of such program must account for **both** possibilities.

• Multiple results are important for modeling e.g. concurrency, as well
as approximating behavior that we do not know
(e.g. random numbers, what the operating system or users do,
what the result of a complex computation is)

Guarded Command Language

assume(F)	- stop execution if F does not hold pretend execution never happened
s1 ; s2	- do first s1, then s2
s1 [] s2	- do either s1 or s2 arbitrarily (drunk if statement)
s*	- execute s zero, once, or more times

Guarded Commands and Relations - Idea

$x = \textit{term}$ ⟹ $\{(x,\textit{term}) \mid \text{true}\}$

gets more complex for more variables

assume(F) ⟹ $\Delta_S = \{(x,x) \mid x \in S\}$

S is set of values for which **F** is true
(satisfying assignments of **F**)

s1 ; s2 ⟹ $r_1 \circ r_2$ (r_1 is for s_1 , r_2 for s_2)

s* ⟹ r* where r describes **s**

s_1 [] s_2 ⟹ $r_1 \cup r_2$ (r_1 is for s_1 , r_2 for s_2)

Relation Composition

$$\begin{tikzcd} a \arrow[r, "r_1"] & b \arrow[r, "r_2"] & c \end{tikzcd}$$

$\rightarrow$ $r_1 \circ r_2 = \{(a,c) \mid \exists\, b.\ \overline{(a,b) \in r_1} \wedge (b,c) \in r_2\}$

$[[\ (x = x + 1 \boxed{;} x = x - 5) \boxed{]]} =$ $\rightarrow$ meaning of

$[[x = x + 1]] \circ [[x = x - 5]] =$

$\{(x,x') \mid x' = x + 1\} \circ \{(x,x') \mid x' = x - 5\} =$

$\{(a,c) \mid \exists b.\ (a,b) \in \{(x,x') \mid x' = x+1\} \wedge$

$(b,c) \in \{(x,x') \mid x' = x-5\}\} =$

$= \{(a,c) \mid \exists b. (b = \underbrace{a+1} \wedge$

$c = b - 5)\} = \{(a,c) \mid c = a + 1 - 5\}$

$= \{(a,c) \mid c = a - 4\}$

Assignment for More Variables

(x, y, z)

var x,y,z

…

y = x + 1;

$z = y + x$

$\{((x,y),(x',y')) \mid x = x' \land y' = x + 1\}$

$\{((x,y,z),(x',y',z')) \mid x = x' \land y' = x + 1 \land z' = z\}$

$\{((x,y,z),(x',y',z')) \mid x' = x \land y' = x + 1 \land z' = 2 \cdot x + 1\}$

‘if’ condition using assume and []

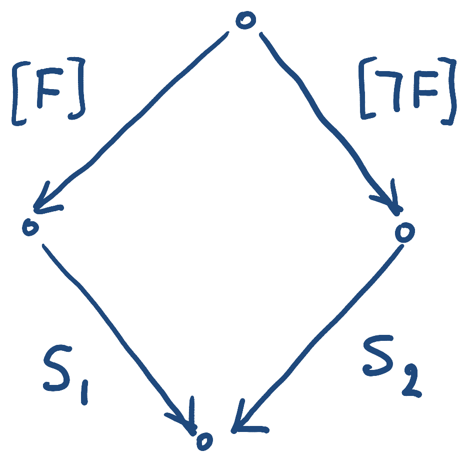
```
if (F)
  s1
else
  s2
```

(assume(F); s1)

[]

(assume(¬F); s2)

CFG:


$$\begin{tikzcd} & \circ \arrow[dl, "{[F]}"'] \arrow[dr, "{[\neg F]}"] & \\ \circ \arrow[dr, "s_1"'] & & \circ \arrow[dl, "s_2"] \\ & \circ & \end{tikzcd}$$

Example: y is absolute value of x

```
if (x>0)
  y = x
else
  y = -x
```

$\{((x,y),(x',y')) \mid \quad \}$

(assume(x>0); y=x)

[]

(assume(¬(x>0)); y=-x)

$[\![\text{assume}(x>0); y=x]\!] = \{ \ldots \mid x>0 \wedge y'=x \wedge x'=x \}$

$[\![\text{assume}(x\leq 0); y=-x]\!] = \{ \ldots \mid x\leq 0 \wedge y'=-x \wedge x'=x \}$

RESULT:

$\{ \qquad \mid (x>0 \wedge y'=x \wedge x'=x) \vee (x\leq 0 \wedge y'=-x \wedge x'=x) \}$

Guards are assume statements

var x

$\underbrace{F \rightarrow}\ c$ shorthand for: assume(F);c

{ (x,x') | R (x,x') }

{(x,x) | F(x) } o {(x,x’) | R(x,x’)} =

{(x,x’) | F(x) /\ R(x,x’) }

Adding a guard F corresponds to adding the condition F to the initial state in the relation.

Analogously:

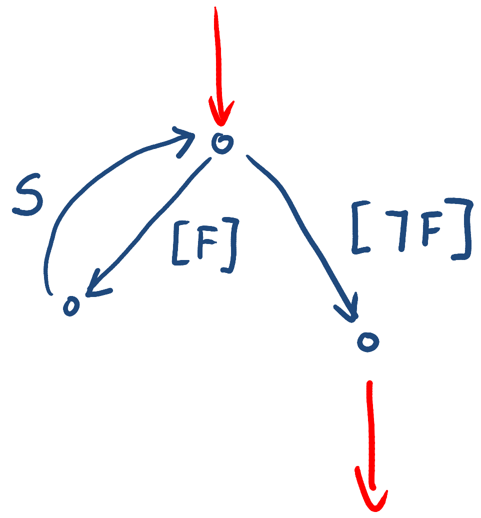
[[c; assume(F)]] = {(x,x’) | R(x,x’) /\ F(x’)}

‘while’ using assume and *

while (F)
  s

(assume(F); s)* ;
assume(¬F)

CFG:



Consider all paths from before the loop to after the loop, and look at the resulting regular expression.

In Control-Flow Graph we denote assume(F) by [F].

x = 3;
x = 4;

Havoc Statement

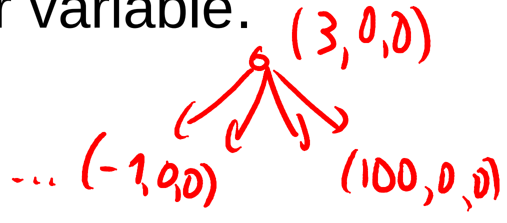
x, y, z

assume(x==3)
assume(x==4)

Havoc Statement

- Havoc statement is another useful declarative statement. It changes a given variable entirely arbitrarily: there will be one possible state for each possible value of integer variable.

$[\![\text{havoc}(x)]\!] = \{((x,y,z),(x',y',z')) \mid y' = y \wedge z' = z\}$



Expressing Assignment with Havoc+Assume

- We can prove that the following equality holds when x does not occur in E:

  x=x+1 havoc(x); assume(x==x+1)

  x = E is havoc(x); assume(x==E)

  In other words, assigning a variable is the same as changing it arbitrarily and then assuming that it has the right value.